VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Důkazové techniky ve výrokové logice

Proofs in Propositional Logic

2013 **Marek Hošťálek**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student:	**Marek Hošťálek**
Studijní program:	B2647 Informační a komunikační technologie
Studijní obor:	2612R025 Informatika a výpočetní technika
Téma:	Důkazové techniky ve výrokové logice Proofs in Propositional Logic

Zásady pro vypracování:

Cílem práce je webová aplikace řešící problematiku dokazování ve výrokové logice. Student nastuduje a naprogramuje sémantické a syntaktické důkazové techniky.

Práce bude obsahovat:
1. Seznámení s jazykem výrokové logiky.
2. Popis sémantických důkazových metod (tabulková metoda, metoda sémantického sporu).
3. Popis syntaktických důkazových metod (rezoluční metoda, přirozená dedukce, systém Hilbertova typu).
4. Webovou aplikaci, která provede důkaz platnosti úsudku ve výrokové logice všemi výše zmíněnými metodami.

Systém bude sloužit jako výukový materiál pro výuku dokazování ve výrokové logice.

Seznam doporučené odborné literatury:

[1] Duží, M.: Logika pro informatiky, Ostrava 2012, VŠB-TU Ostrava, ISBN: 978-80-248-2662-2
[2] Švejdar, V.: Logika - neúplnost, složitost a nutnost, ACADEMIA 2002, ISBN: 80-200-1005-X

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Mgr. Marek Menšík, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne: 18.7. 2013

……..……
podpis studenta

Poděkování

Rád bych poděkoval Mgr. Marku Menšíkovi, Ph.D. za odbornou pomoc a konzultaci při vytváření této bakalářské práce.

Prohlášení zástupce spolupracující právnické nebo fyzické osoby

„Souhlasím se zveřejněním této bakalářské/diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.“

Dne: 18.7. 2013 ……..……

podpis zástupce

Abstrakt

Systém, který bude výstupem této práce, bude sloužit pro dokazování platných úsudků výrokové logiky. Důkazové metody použité v této práci jsou důkazy tabulkovou metodou, sémantickým sporem, rezoluční metodou, přirozenou dedukcí a systém Hilbertova typu. Práce obsahuje jazyk výrokové logiky, teorii důkazových metod, popis použitých technologií a uživatelskou příručku.

Klíčová slova

Výroková logika, tabulková metoda, sémantický spor, rezoluční metoda, přirozená dedukce, Hilbertův princip, JavaFX, návrhový vzor Kompozit, návrhový vzor Builder.

Abstract

The system, which will be the outcome of this work, will serve as proof of valid proofs of propositional logic. Proof methods used in this work are table method, semantic contradiction, resolution method, natural deduction and Hilbert principle. The work contains language of propositional logic, theory proofs methods, a details of the used technology and user manual.

Key words

Propositional logic, table method, semantic contradiction, resolution method, nature deduction, Hilbertův principle, JavaFX, design pattern Composite, design pattern Builder.

Seznam použitých zkratek

Zkratka	Anglický význam	Český význam
API	Application Programming Interface	Rozhraní pro programování aplikací
CDDL	Common Development and Distribution License	
CSS	Cascading Style Sheets	Kaskádové styly
DUF		Dobře utvořená formule
GPL	General Public License	Všeobecná veřejná licence
HTTP	Hypertext Transfer Protocol	
JRE	Java Runtime Environment	
KNF	Conjunctive Normal Form	Konjunktivní normální forma
OS	Operating System	Operační systém
RIA	Rich Internet Applications	Bohaté internetové aplikace
XML	Extensible Markup Language	Rozšiřitelný značkovací jazyk

Obsah

1 Úvod

Cílem mé bakalářské práce je systém, který bude řešit dokazování ve výrokové logice, budou zde rozebrány důkazové metody a jazyk výrokové logiky.

Práce se skládá z praktické a teoretické části. Výstupem praktické části je webová aplikace. V teoretické části popisuji jazyk výrokové logiky, prostředky, které využiji pro tvorbu webové aplikace a popis důkazových technik. Jedná se o tabulkovou metodu, metodu sémantického sporu, rezoluční metodu, přirozenou dedukci a Hilbertův princip.

Pomocí aplikace si budou moci studenti kurzů zaměřených na dokazování platnosti úsudku ve výrokové logice ověřit správnost svých výsledků.

Výsledkem bakalářské práce bude již zmiňovaná webová aplikace a teoretický materiál, který může sloužit pro nastudování jazyka výrokové logiky a dokazovacích technik výrokové logiky.

1.1 Struktura práce

V následující kapitole je uvedena teorie. V teoretické části je uveden jazyk výrokové logiky a popsány dokazovací metody, s kterými bude systém pracovat.

Třetí kapitola popisuje využité návrhové vzory. Platformu, v které je aplikace vyvíjena, popis vývojového prostředí, popis programových balíčků a uživatelskou příručku.

Poslední kapitola je věnována závěru.

2 Teorie

2.1 Jazyk výrokové logiky

V této kapitole čerpám z [1].

Výroková logika analyzuje věty až do úrovně elementárních výroků. Strukturu těchto elementárních výroků již dále nezkoumá. Přitom

Výrok je tvrzení, o němž má smysl prohlásit, zda je pravdivé či nepravdivé.

Tato ”definice” se zdá být až banální, pokud si neuvědomíme, že ne každá věta vyjadřuje výrok. Např. věta *Francouzský král je holohlavý* nemůže být v současné době (kdy neexistuje francouzský král) ani pravdivá, ani nepravdivá. Pravdivá zřejmě není. Kdyby byla nepravdivá, pak by bylo pravda, že francouzský král není holohlavý, což rovněž není možné, protože francouzský král neexistuje. Tedy kdyby nastal jeden z těchto případů (pravdivost či nepravdivost), vyplývala by z této věty existence francouzského krále! Klasická výroková logika tedy ctí princip dvojhodnotovosti (tercium non datur – Chrisipos ze Solov).

Výroky dělíme na jednoduché a složené. Elementární (jednoduchý) výrok je takové tvrzení, jehož žádná vlastní část již není výrokem. Složený výrok pak má vlastní části – výroky. Výroková logika zkoumá strukturu těchto složených výroků v tom smyslu, že zkoumá způsob skládání jednoduchých výroků do složených pomocí logických spojek. Je to tedy teorie logických spojek. Přitom ovšem zachovává žádoucí princip skladebnosti (kompozicionality), podle něhož je pravdivostní hodnota složeného výroku jednoznačně určena jen pravdivostními hodnotami jeho složek a povahou spojení těchto složek (tj. logickou povahou spojek).

Příklad. Složené výroky.

V Praze prší a v Brně je hezky.
| | |
el. výrok | el. Výrok
spojka

Není pravda, že v Praze prší.
| |
Spojka el. Výrok

Jazyk výrokové logiky musí proto obsahovat symboly zastupující jednotlivé elementární výroky, tzv. výrokové symboly (proměnné), které budou nabývat hodnot pravda, nepravda, symboly pro logické spojky a případné pomocné symboly. Každý jazyk je nejlépe definován abecedou a gramatikou. Abeceda určuje typy symbolů, které mohou být v jazyce použity, gramatika pak zadává pravidla tvorby (nekonečně mnoha) dobře utvořených výrazů, tj. v tomto popřípadě formulí.

Abeceda jazyka výrokové logiky je množina následujících symbolů:

- Výrokové symboly: *p*, *q*, *r*, ... (případně s indexy)
- Symboly logických spojek (funktorů): ¬, v, ∧, ↄ, ≡
- Pomocné symboly (závorky): (,), případně [,],{,}

Symboly ¬, ∨, ∧, ↄ, ≡ nazýváme po řadě spojky negace, disjunkce, konjunkce, implikace, ekvivalence.

Gramatika jazyka výrokové logiky rekurzivně definuje nekonečnou množinu formulí:

1) Výrokové symboly jsou formule (báze definice).
2) Jsou-li výrazy *A*, *B* formule, pak jsou formulemi i výrazy
   (¬A,) (A ∧ B),(A ∨ B),(A ↄ B),(A ≡ B) (*)
3) Jiných formulí výrokové logiky, než podle bodů (1), (2) není (uzávěr definice).

Jazyk výrokové logiky je množina všech formulí výrokové logiky.

Formule vzniklé podle bodu (1) nazýváme elementárními (atomárními, primitivními) formulemi, formule vzniklé podle bodu (2) složenými formulemi. Formule A, B jsou bezprostředními podformulemi formulí (*). Maximální počet do sebe vnořených závorkových dvojic (,) vyskytujících se ve formuli udává (hierarchický) řád formule.

2.2 Tabulková metoda

V této kapitole čerpám z [2][6].

Řekněme, že máme nějakou formuli výrokové logiky ϕ, ve které vyskytuji nějaká atomická tvrzeni p1, p2, . . . , pn a žádna další atomická tvrzeni Jako přiklad si vezměme třeba formuli

$$p \wedge (q \supset \neg r)$$

ve které se vyskytuji atomická tvrzeni p, q a r. Pokud bychom u každého z tvrzeni p, q, r věděli,jestli je toto tvrzeni pravdivé nebo nepravdivé (tj. jestli má pravdivostní hodnotu 1 nebo 0), mohli bychom určit pravdivostní hodnotu celého tvrzeni na základě dříve uvedených tabulek pravdivostních hodnot pro jednotlivé logické spojky.

Řekněme, že p platí, q neplatí a r platí, takže p a r mají pravdivostní hodnotu 1 a q pravdivostní hodnotu 0. Podle tabulky pro negaci má ¬r pravdivostní hodnotu 0 (protože r má pravdivostní hodnotu 1). Tvrzeni q má pravdivostní hodnotu 0 a tvrzeni ¬r má pravdivostní hodnotu 0, takže podle tabulky pro implikaci má q ↄ ¬r pravdivostní hodnotu 1. Protože p má pravdivostní hodnotu 1 a (q ↄ ¬r) také pravdivostní hodnotu 1, má podle tabulky pro konjunkci p ∧ (q ↄ ¬r) pravdivostní hodnotu 1.

Shrňme kroky z předchozího odstavce v následující tabulce:

	Tvrzení	Pravdivostní hodnota	Komentář
1.	p	1	předpoklad
2.	q	0	předpoklad
3.	r	1	předpoklad
4.	¬r	0	podle řádku 3 a tabulky pro ¬
5.	q ↄ ¬r	1	podle řádků 2 a 4 a tabulky pro →
6.	p ∧ (q ↄ ¬r)	1	podle řádků 1 a 5 a tabulky pro ∧

Tabulka.2.1: Pravdivostní hodnoty formule p ∧ (q ↄ ¬r)

Vynecháme číslováni a komentáře a napíšeme tutéž tabulku „naležato“ tj. tak, že sloupce budou odpovídat podformulím formule p ∧ (q ↄ ¬q), horní řádek bude obsahovat jednotlivé podformule a řádek pod nim pravdivostní hodnoty těchto podformulí:

p	q	r	¬r	q ↄ ¬r	p ∧(q ↄ ¬r)
1	0	1	0	1	1

Tabulka.2.2: Pravdivostní hodnoty formule, bez komentářů.

Poté, není problém doplnit do tabulky další řádky odpovídající všem dalším možnostem, jak mohou atomická tvrzeni p, q a r platit nebo neplatit:

p	q	r	¬r	q ↄ ¬r	p ∧ (q ↄ ¬r)
0	0	0	1	1	0
0	0	1	0	1	0
0	1	0	1	1	0
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0

Tabulka.2.3: Kompletní tabulka pro všechny pravdivostní hodnoty.

Vzhledem k tomu, že pravdivostní hodnota formule výrokové logiky nezávisí na pravdivostních hodnotách atomických tvrzeni, která se v ni nevyskytuji, dává nám taková tabulka úplný přehled o tom, jakých pravdivostních hodnot nabývá daná formule při jednotlivých pravdivostních ohodnoceních (tj. při jednotlivých interpretacích).

Pokud například uvidíme, že ve sloupci tabulky odpovídající formuli ϕ jsou samé hodnoty 1, tj. že ve všech interpretacích je formule pravdivá, můžeme z toho hned usoudit, že formule ϕ je tautologie.

Podobně pokud uvidíme ve sloupečku formule ϕ samé hodnoty 0, můžeme z toho usoudit, že ϕ je kontradikce. Pokud sloupeček obsahuje alespoň jednu hodnotu 1, můžeme zase říct, že formule ϕ je splnitelná.

Obecně se při použiti tabulkové metody nemusíme omezovat jen na výrokovou logiku nebo jen na to, že zkoušíme všechna možná přiřazeni pravdivostních hodnot atomickým tvrzením. Pokud je například nějaká formule ϕ sestavena z nějakých menších formuli A1,A2, . . . ,An pomoci logických spojek, můžeme v tabulce vyzkoušet všechna pravdivostní ohodnoceni podformulí A1,A2, . . . ,An, tj. s podformulemi A1,A2, . . . ,An, můžeme zacházet podobně jako s atomickými tvrzeními. Pokud ve sloupci formule ϕ vyjdou samé hodnoty 1, opět můžeme oprávněně prohlásit, že ϕ je tautologie, a podobně, pokud vyjdou samé hodnoty 0, můžeme oprávněně prohlásit, že ϕ je kontradikce.

Příklad:

p	q	r	$p \supset q$	$\neg q$	$r \vee \neg q$	$\neg r$	$\neg p$
0	0	0	1	1	1	1	1
1	0	0	0	1	1	1	0
0	1	0	1	0	0	1	1
1	1	0	1	0	0	1	0
0	0	1	1	1	1	0	1
1	0	1	1	1	1	0	0
0	1	1	1	0	1	0	1
1	1	1	1	0	1	0	0

Tabulka.2.4: Příklad důkazu tabulkou

Příklad:

a	b	c	b∧c	as ↄ(b∧c)	¬c	b∧¬c	¬a
0	0	0	0	1	1	0	1
1	0	0	0	0	1	0	0
0	1	0	0	1	1	1	1
1	1	0	0	0	1	1	0
0	0	1	0	1	0	0	1
1	0	1	0	0	0	0	0
0	1	1	1	1	0	0	1
1	1	1	1	1	0	0	0

Tabulka.2.5: Příklad důkazu tabulkou

2.3 Sémantický spor

V této kapitole čerpám z [1].

Chceme-li ověřit platnost úsudku sporem, předpokládáme, že úsudek platný není. Dle definice je úsudek neplatný, jestliže existuje ohodnocení výrokových proměnných vyskytujících se v předpokladech a závěru takové, že jsou v něm všechny předpoklady pravdivé a závěr nepravdivý. Jinými slovy, úsudek je neplatný, jestliže existuje model množiny předpokladů ve kterém je závěr nepravdivý.

Příklad:

Nyní ověříme, zda formule $\neg p$ logicky vyplývá z množiny $\{p \supset q, r \vee \neg q, \neg r\}$. Názorně tedy prověříme úsudkové schéma

$p \supset q, r \vee \neg q, \neg r \; / \; \neg p$
1 1 1 0
0 1
1 1 0 1
|_____0 spor !
Úsudek je platný.

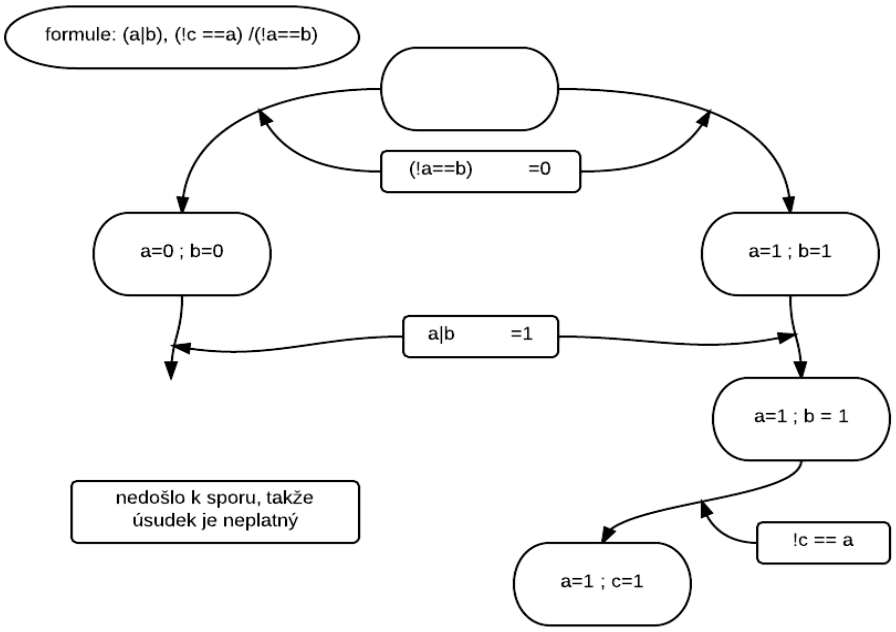
Příklad:

as ɔ(b∧c), ¬b∧c / ¬a
1 1 0
1 1 0 1 1
1 1
|_____spor s b
Úsudek je platný.

Příklad:

[(p∧ q) ∨ r] ɔ ss ,¬s, q / p ∨ r
1 1 1 0
0 0 0
0 0
0 1 0 0
nedošlo k sporu, úsudek je neplatný

Řešení metodou sémantického sporu jsem se rozhodl znázornit skrz stromovou strukturu. Vycházím z toho, že předpoklady odpovídají závěru. To znamená, že pro metodu sporu ohodnotím předpoklady jako pravdivé a závěr jako bude negovaný tzn. nepravdivý. Aby byl úsudek platný musím v průběhu procházení výrazu najít takové ohodnocení předpokladů, v kterém i dojde k sporu s ohodnocením závěru. V kořenu stromu bude znázorněn negovaný závěr a v uzlech pod ním budou znázorněny pravdivé předpoklady.



Obrázek 2 .1: *Ohodnocení sémantického sporu*

2.4 Rezoluční metoda

V této kapitole čerpám z [1].

Další důležitou metodou logického vyplývání, řešení úlohy – co vyplývá z daných předpokladů, apod. je tzv. metoda základní rezoluce. Tato metoda je uplatnitelná na formule v konjunktivní normální formě (KNF).

Obecně, jak se dovíme v kapitole 3, je rezoluční metoda důkaz sporem. Ovšem ve výrokové logice ji můžeme použít také pro přímý důkaz. Je tomu tak proto, že rezoluční pravidlo zachovává pravdivost. Tedy rezolventa, kterou z daných předpokladů odvodíme, z nich vyplývá. A nyní přesněji:

Rezoluční pravidlo odvozování: Nechť l je literál. Z formule $(A \vee l) \wedge (B \vee \neg l)$ odvoď formuli $(A \vee B)$. Zapisujeme:

$$\frac{(A \vee l) \wedge (B \vee \neg l)}{(A \vee B)}$$

Toto pravidlo není přechodem k ekvivalentní formuli, ale zachovává *pravdivost* tedy rezolventa z daných předpokladů vyplývá**.** Musíme si uvědomit, že někdy je nutné přidat mezi množinu předpokladů axiomy, v případě přímého důkazu rezoluční metodou.

Důkaz: Nechť je formule $(A \vee l) \wedge (B \vee \neg l)$ pravdivá při nějaké valuaci v. Pak při této valuaci musí být pravdivé oba disjunkty (tzv. klausule) $(A \vee l)$ a $(B \vee \neg l)$. Nechť je dále $v(l) = 0$. Pak $w(A) = 1$ a tedy $w(A \vee B) = 1$. Nechť je naopak $v(l) = 1$. Pak $w(\neg l) = 0$ a musí být $w(B) = 1$, a tedy $w(A \vee B) = 1$. V obou případech je tedy formule $(A \vee B)$ pravdivá v modelu původní formule, a tedy je pravdivá v každém modelu předpokladů, neboť jsme zkoumali *libovolnou* valuaci v:

$$\boldsymbol{(A \vee l) \wedge (B \vee \neg l) \models (A \vee B).}$$

To nám poskytuje návod, jak řešit úlohu, co vyplývá z dané formule, resp. množiny formulí.
Postup řešení:

Pozn.: Jednotlivé disjunkty v KNF nazýváme klausule, a proto je KNF také nazývána klausulární forma.

a) Nepřímý důkaz, že formule A je tautologie: Formuli A znegujeme a převedeme do KNF. Nyní uplatňujeme pravidlo rezoluce. Pokud při postupném ”vyškrtávání” literálů s opačným znaménkem dospějeme k prázdné klausuli, je tato evidentně nesplnitelná, tedy také původní ¬A je nesplnitelná a A je tautologie.

b) Nepřímý důkaz správnosti úsudku $P_1,...,P_n \models Z$. Závěr Z znegujeme a dokazujeme, že množina $\{P_1,...,P_n, \neg Z\}$ je sporná. Jinými slovy, dokazujeme, že formule $(P_1 \wedge ... \wedge P_n) \supset Z$ je tautologie, tedy že její negace $P_1 \wedge ... \wedge P_n \wedge \neg Z$ je kontradikce.

Příklad:

Ověříme platnost úsudku $p \supset q$, $r \vee \neg q$, $\neg r$ / $\neg p$ nepřímým důkazem. Jednotlivé klausule zapíšeme pod sebe (s negovaným závěrem) a uplatňujeme pravidlo rezoluce:

1. $\neg p \vee q$
2. $r \vee \neg q$
3. $\neg r$
4. p negovaný závěr

---

5. q (1. a 4)
6. r (2. a 5.)
7. (3. a 6.)

Dostali jsme prázdnou klausuli, která je nesplnitelná. Tedy negovaný závěr je ve sporu s předpoklady, proto je úsudek platný.

1. $q \vee p$
2. $p \vee r$
3. $\neg q \vee \neg r$
4. $\neg p$ negovaný závěr

---

5. $p \vee \neg r$ (1. a 3.)
6. p (2. a 5.)
7. (4. a 6.)

Nalezen spor. Úsudek je platný.

1. $\neg p \vee q$
2. $p \vee r$
3. $\neg q$
4. $\neg r$ negovaný závěr

---

5. $q \vee r$ (1. a 2.)
6. r (3. a 5.)
7. (4. a 6.)

Nalezen spor. Úsudek je platný.

Řešení rezoluční metodou budeme provádět v aplikaci pomocí nepřímého důkazu, to znamená, že budu předpokládat, že předpoklady jsou pravdivé a závěr ne (negovaný závěr).

2.5 Přirozená dedukce

V této kapitole čerpám z [1].

Přirozená dedukce je metoda výstavby formálního systému tzn. důkazového kalkulu logiky. Formální systémy logiky je možno v zásadě rozdělit na systémy axiomatické a předpokladové. Předpokladový systém je právě přirozená dedukce v alternativě polské, né však gentzenovské. Formální systém je postaven zásadně na syntaktické bázi, podobně jako je to u rezoluční metody. To znamená, že jazyk logiky uvažujeme neinterpretovaný a veškeré manipulace s ním jsou výhradně syntaktické, na základě odvozovacích pravidel. Takový souhrn nazýváme také logický kalkul.

V tomto případě se formální systém sestavuje z dvou složek, a to

· *jazyk* – z jeho symbolů vytváříme konečné posloupnosti – formule (kterým zde nepřisuzujeme žádný smysl)

· *odvozovací pravidla* – operace na formulích, které umožňují otestování "platnosti výroků" prostřednictvím konstrukce důkazu.

Cílem tohoto postupu je získat v rámci formálního systému jistou jeho část – formální teorii jako souhrn dokazatelných formulí – ***teorémů.*** Interpretace formální teorie (která není součástí formálního systému) dodává teorii význam a činí ji vhodnou pro aplikace v prokazování úsudku.

Systém přirozené dedukce vychází z několika jednoduchých dedukčních (odvozovacích) pravidel, která se považují za výchozí a která se proto nemusí dokazovat. Na základě těchto výchozích pravidel se pak dokazují další obtížnější dedukční pravidla.

Dedukční pravidla s nulovým počtem předpokladů jsou tzv. *axiómy* formálního systému (obdoby tautologií ze sémantického pojetí výrokové logiky). Jako axiómy zde používáme formule tvaru $A \vee \neg A$, popř. $A \supset A$. Pro dobře definovaný *korektní* formální systém (výrokové) logiky platí, že množina teorémů je stejná s množinou tautologií, tedy že axiómy jsou logicky pravdivé a odvozovací pravidla zachovávají pravdivost. Dedukční pravidlo má obecně následující tvar:

$$F_1, F_2,...,F_m \vdash G_1, G_2,...,G_n.$$

Pravidlo "interpretujeme" takto: ze současné platnosti všech formulí F_1, $F_2,...,F_m$ (předpokladů) plyne platnost libovolné z formulí G_1, $G_2,...,G_n$. Byly-li dokázány všechny formule z levé strany dedukčního pravidla, pak můžeme považovat za dokázány i libovolné formule z pravé strany pravidla.

Výchozími (nedokazovanými) dedukčními pravidly jsou:

Axiómy	$\vdash A \vee \neg A$, $\vdash A \supset A$	
Zavedení konjunkce	$A, B \vdash A \wedge B$	ZK
Eliminace konjunkce	$A \wedge B \vdash A, B$	EK
Zavedení disjunkce	$A \vdash A \vee B$ nebo $B \vdash A \vee B$	ZD
Eliminace disjunkce	$A \vee B, \neg A \vdash B$ nebo $A \vee B, \neg B \vdash A$	ED
Zavedení implikace	$B \vdash A \supset B$	ZI
Eliminace implikace	$A \supset B, A \vdash B$	EI modus ponens MP
Zavedení ekvivalence	$A \supset B, B \supset A \vdash A \equiv B$	ZE
Eliminace ekvivalence	$A \equiv B \vdash A \supset B, B \supset A$	EE

Tabulka.2.6: Tabulka s dedukčními pravidly

Uvedená pravidla ve svém souhrnu charakterizují význam funktorů ¬,∧, ∨,⊃,≡.

Dále v aplikaci využívám již dříve dokázaná pravidla, která jsou platná, proto je není nutné dokazovat zvlášť samostatně.

Zavedení negace	$A \vdash \neg\neg A$	ZN
Eliminace negace	$\neg\neg A \vdash A$	EN
Negace disjunkce	$\neg(A \vee B) \vdash \neg A \wedge \neg B$	ND (de Morganův zákon)
Negace konjunkce	$\neg(A \wedge B) \vdash \neg A \vee \neg B$	NK (de Morganův zákon)
Negace implikace	$\neg(A \supset B) \vdash A \wedge \neg B$	NI
Tranzitivita implikace	$A \supset B, B \supset C \vdash A \supset C$	TI
Transpozice	$A \supset B \vdash \neg B \supset \neg A$	TR
Modus tollens	$A \supset B, \neg B \vdash \neg A$	MT

Tabulka.2.7: Tabulka s dříve dokázanými pravidly

Příklad:

Důkaz (nepřímý):
1. $p \supset q$ předpoklad
2. $\neg q$ předpoklad
3. p předpoklad nepřímého důkazu
4. q MP: 1,3 spor s 2

Na výše uvedeném příkladu došlo k sporu s negovaným závěrem tzn. že úsudek je platný (nalezl jsem spor). V aplikaci budu využívat také nepřímého důkazu (důkazu sporem).

Příklad:

Důkaz (nepřímý):

1. ¬As ↄ¬B	předpoklad
2. B	předpoklad
3. ¬A	negovaný závěr
4. ¬B	EI: 1,3 spor s 2

Úsudek je platný.

Příklad:

Důkaz (přímý):

As ↄB, B ↄ C, A |- C

1. As ↄB	předpoklad
2. B ↄ C	předpoklad
3. A	předpoklad
4. B	EI: 1,3
5. C	EI: 2,4

Byl nalezen závěr, úsudek je platný

2.6 Hilbertův princip

V této kapitole čerpám z [1].

Formální axiomatický systém libovolné teorie (a speciálně také výrokové logiky) je zadán touto trojicí údajů:

1. jazykem,
2. množinou axiómů,
3. množinou odvozovacích pravidel.

Jazyk teorie je množina všech (dobře utvořených) formulí(DUF) jazyka. Množina axiómů teorie je vybraná podmnožina množiny všech formulí. Axiómy představují základní teorémy teorie, které se považují za výchozí. Odvozovací pravidla umožňují odvozovat (dokazovat) nové teorémy na základě axiómů a teorémů již dokázaných.

Množina axiómů je vždy neprázdná a musí být rozhodnutelná v množině DUF (jinak bychom nemohli v takovém systému nic dokazovat). To znamená, že existuje algoritmus, který pro každou DUF určí, zda je to axióm nebo ne. Může být konečná nebo nekonečná. Konečná množina axiómů je triviálně rozhodnutelná. Nekonečné množiny axiómů musí být charakterizovány algoritmem vytváření axiómů, nebo častěji konečnou množinou tzv. axiomových schémat. Axiómy jsou voleny tak, aby byly pravdivé v každé interpretaci – tautologie. Navíc stanovujeme tzv. speciální axiómy, které

charakterizují přímo danou teorii (např. aritmetiku přirozených čísel), a ty volíme tak, aby byly pravdivé v zamýšlené interpretaci teorie. (Výroková logika či predikátová logika 1. Řádu – mohou být tedy považovány za teorie bez speciálních axiómů – logické důkazové kalkuly.)

Množina odvozovacích pravidel je tvořena několika nebo dokonce jen jedním pravidlem (jsou-li axiómy reprezentovány schématy). Jak jsme viděli v předchozí kapitole, systém přirozené dedukce pracuje pouze se dvěma axiómy, ale zato s podstatně větším počtem dedukčních pravidel. Odvozovací pravidla převádějí DUF na DUF a jsou volena tak, aby byla sémanticky korektní, tj. aby "zachovávala pravdivost" (jinak bychom obdrželi nekorektní systém, ve kterém je možno dokázat vše, a takový systém jistě není z praktického hlediska užitečný). Odvozovací pravidla tedy umožňují vytvářet teorémy, tj. dokazatelné formule. Důkaz je konečná posloupnost kroků – DUF, z nichž každá je buď axióm nebo vznikne z předchozích DUF pomocí odvozovacího pravidla. Posledním krokem je dokazovaná formule – teorém.

Někdy bývá stanoven ještě jeden přirozený "kosmetický" požadavek na množinu axiómů: Množina axiómů má být nezávislá, tj. minimální v tom smyslu, že žádný axióm není dokazatelný z ostatních axiómů.

Axiomatických systémů neboli důkazových kalkulů výrokové a predikátové logiky je vytvořeno velké množství. Liší se navzájem jazykem, množinou axiómů a odvozovacími pravidly. Všechny však představují jen různé formalizace "intuitivní logiky". Všechny formalizace mají společnou vlastnost: Jsou korektní (každá dokazatelná formule, tj. logický teorém důkazového kalkulu, musí být tautologií). V tomto smyslu jsou všechny formalizace ekvivalentní.

Jazyk pro Hilbertův systém ve výrokové logice:

Abeceda:

Výrokové symboly: p, q, r,... (případně s indexy)
Logické funktory: ¬, ↄ
Závorky: (,) (/případně [,],{,})

Gramatika (DUF):

p, q, r,... jsou formule.
Je-li A formule, pak $(\neg A)$ je formule.
Jsou-li A, B formule, pak $(A \supset B)$ je formule.
Jiných formulí než podle (1), (2), (3) není.

Jazyk: množina všech (dobře utvořených) formulí.

Axiomová schémata:

A1: A ↄ (B ↄ A)
A2: (A ↄ (B ↄ C)) ↄ ((A ↄ B) ↄ (A ↄ C))
A3: (¬B ↄ ¬A) ↄ (A ↄ B)

Odvozovací pravidlo: Modus ponens (MP): A, A ⊃ B |– B

A, *B* nejsou formulemi, ale metasymboly sloužícími k označení formulí. Každé axiomové schéma označuje nekonečnou třídu axiómů daného tvaru. Kdybychom axiomová schémata nahradili axiómy

1. $p \supset (q \supset p)$
2. $(p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$
3. $(\neg q \supset \neg p) \supset (p \supset q)$

museli bychom rozšířit množinu odvozovacích pravidel o další pravidlo, tzv. pravidlo substituce, abychom získali ekvivalentní důkazový kalkul. Pravidlo substituce zní: Dosadíme-li v dokázané formuli za jednotlivé výrokové symboly jakékoliv jiné formule (za každý výskyt téhož výrokového symbolu vždy tutéž formuli), pak získáme opět dokázanou formuli (teorém).

Definovaný axiomatický systém pracuje pouze s funktory ¬, ⊃. Vzhledem k tomu, že pravdivostní funkce příslušné k těmto funktorům tvoří funkcionálně úplný systém, postačí tyto funktory k vytvoření sémanticky úplné logiky. Ostatní výrokotvorné funktory můžeme používat jako zkratky (zkracující a zpřehledňující zápis formulí) definované takto:

$A \wedge B =_{df} \neg (A \supset \neg B)$
$A \vee B =_{df} \neg A \supset B$
$A \equiv B =_{df} (A \supset B) \wedge (B \supset A)$

Symboly ∧, ∨, ≡ nepatří do jazyka definovaného axiomatického systému, jsou to metasymboly sloužící k označování složených formulí jistého typu.

Příklad:

Důkaz formule $A \supset C$ za předpokladů $A \supset B$, $B \supset C$:

1. $A \supset B$	1. předpoklad
2. B ⊃ C	2. předpoklad
3. $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$	ax. A2
4. $(B \supset C) \supset (A \supset (B \supset C))$	ax. A1 $A/(B \supset C)$, B/A
5. $A \supset (B \supset C)$	MP:2,4
6. $(A \supset B) \supset (A \supset C)$	MP:5,3
7. $A \supset C$	MP:1,6

Pomocí pravidla modus ponens a využití axiomů jsem došel k závěru. Formule je platná.

Příklad:

1. *A*	1. předpoklad
2. ¬*A*	2. předpoklad
3.(¬B ↄ ¬A) ↄ (As ↄB)	ax. A3
4. ¬As ↄ (¬B ↄ ¬A)	ax. A1
5. ¬B ↄ ¬A	MP: 2,4
6. As ↄB	MP: 5,3
7. B	MP: 1,6

Pomocí pravidla modus ponens a využití axiomů jsem došel k závěru. Formule je platná.

V aplikaci provádím Hilbertův princip pomocí aplikace axiomu č. 1, protože při aplikaci všech axiomů na každý předpoklad dojde k velké explozi formulí, které program vygeneruje, proto používám pouze axiom č. 1 a rozšiřuji s ním každou formuli, po rozšíření uplatňuji pravidlo modus ponnes. Bohužel jsem nepřišel na jiné řešení, které by bylo aplikovatelné v reálném čase. U testovaných příkladů jsem nalezl řešení.

3 Případová studie

V této kapitole jsou popsány prvky, které využívám při vývoji aplikace.

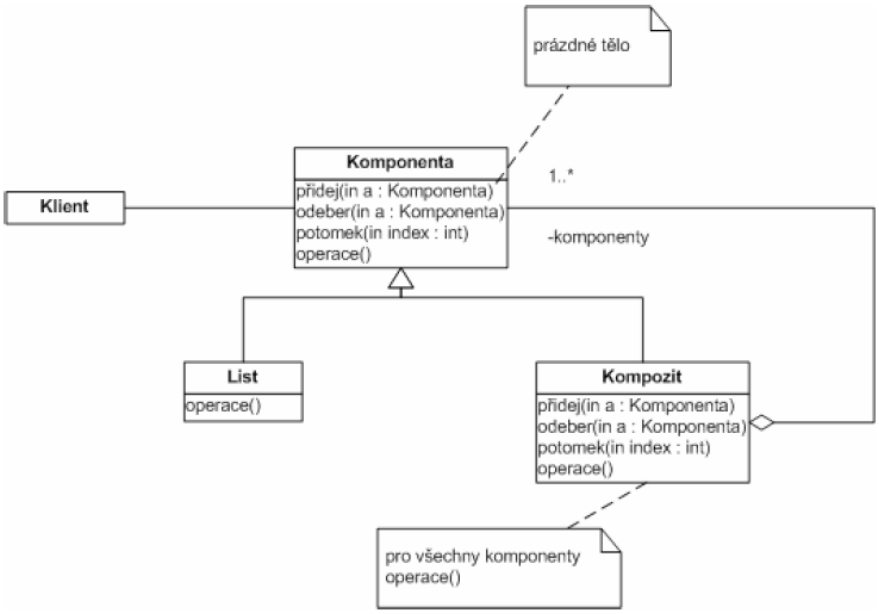
3.1 Návrhové vzory

3.1.1 Kompozit

Předpokládám, že pro uložení logického výrazu budu využívat stromové struktury, proto se budu snažit využívat návrhového vzoru Kompozit.

Návrhový vzor Kompozit, patří do skupiny tzv. strukturálních návrhových vzorů. Umožňuje složení objektů do stromové struktury reprezentující hierarchii celků a jejich částí. Navíc tento návrhový vzor umožňuje klientům pracovat s celky i jejich částmi jednotným způsobem. Klient tedy nemusí rozlišovat, zda-li se jedná o celek nebo část.

Strukturování do stromu je natolik často používanou metodou, že je nutné provést zobecnění a definovat vlastní návrhový vzor Kompozit . Jádrem je třída Komponenta, která je konkretizována do nějaké dále nedělitelné třídy List nebo do třídy Kompozit sloužící jako nástroj pro skládání dalších komponent. Rekurzivně tak můžeme vytvořit hierarchii teoreticky neomezené hloubky. Za povšimnutí stojí i společná *operace* proveditelná jak listem tak kompozitem. Její provedení v případě instance třídy Kompozit spočívá v tom, že tato instance přepošle tuto zprávu na všechny své části, ze kterých je složena.



Obrázek 3 .1: *Návrhový vzor Kompozit*

Tento návrhový vzor je používán zejména v návrhu grafického uživatelského rozhraní, kde kompozitem jsou plochy (např. to může být samotné okno či dialog), do kterých jsou vkládány další plochy nebo atomické prvky jako např. textová pole, tlačítka či posuvníky. Vytváří se tak hierarchie objektů uživatelské prostředí, které mají např. společnou operaci zobrazení vykreslující jejich obsah na obrazovku. Plochy přepošlou tento požadavek na všechny své komponenty, zatímco atomické prvky se vykreslí podle jim daného algoritmu.

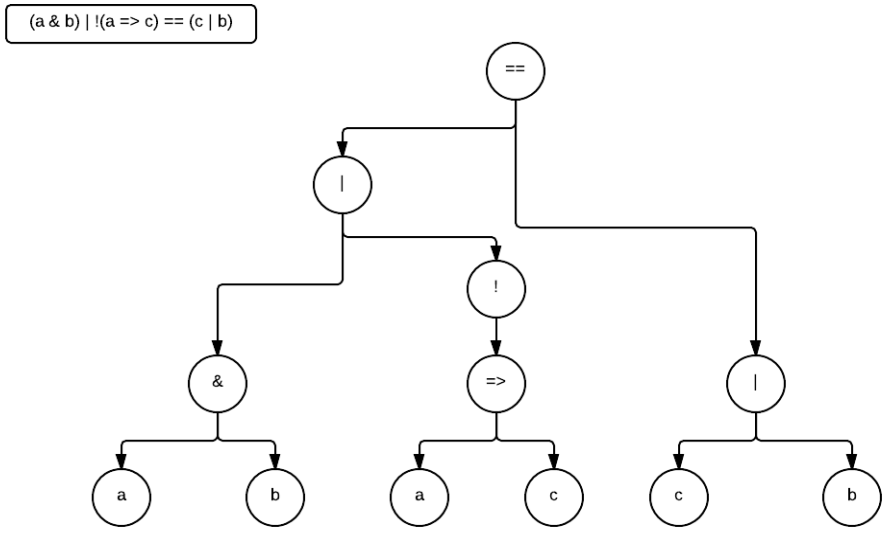
Zdroj: [3][4]

Ve své aplikaci předpokládám, že budu využívat návrhového vzoru kompozit pro uložení logického výrazu do stromové struktury. Priorita logických operací je (1-nejvyšší, 5-nejnižší):

název operace	*priorita*	*značení*
negace	1.	!, ¬
disjunkce	2.	|, ∨
konjunkce	3.	&, ∧
implikace	4.	=>, ↄ
ekvivalence	5.	= =, ≡

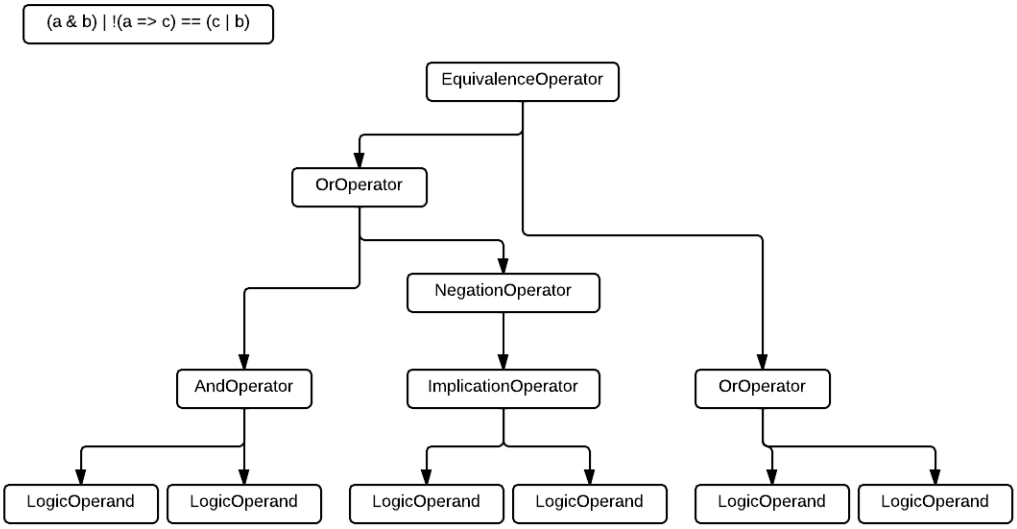
Tabulka.3.1: Priority logickáých operací

Na obrázku vidíme ukázku logického výrazu a jeho uspořádání do stromové struktury, dle priority logických operací. Operace s nejvyšší prioritou se vykonávají jako první a jsou tak nejblíže listům stromu.



Obrázek 3 .2: *Uspořádání do stromové struktury*

Na obrázku vidíme interpretaci tříd logického výrazu do stromu. Na pozici listů jsou operandy logického výrazu. Na místě uzlů se pak nachází třídy, které řeší jednotlivé logické operace.



Obrázek 3 .3: *Interpretace tříd do stromu*

3.1.2 **Builder**

Pro vytvoření stromové struktury budu používat Builder, který se bude řídit návrhovým vzorem Builder.

Návrhový vzor stavitel (anglicky Builder) je softwarový návrhový vzor. Patří do skupiny vytvářejících (creational) návrhových vzorů.

Slouží k abstrahování tvorby složitých objektů. A to tak, aby stejné výrobní schéma mohlo být použito pro tvorbu různých objektů. Způsob konstrukce těchto objektů je tedy stejný, jednotlivé kroky se však liší. Často se používá společně s návrhovým vzorem Kompozit (Composite), konkrétně vytvářené objekty mohou být typu kompozit, dle zmíněného vzoru.

Definice:

„Návrhový vzor oddělující konstrukci složitých objektů od jejich reprezentace. Čímž je možné použít stejný proces konstrukce pro rozdílné reprezentace.“

Stručná historie:

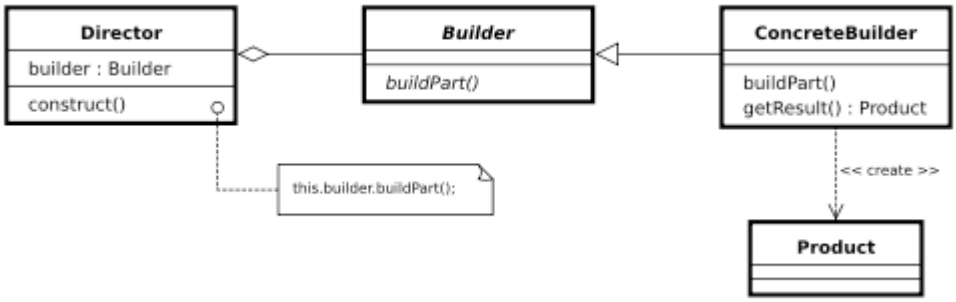
První publikace popisující myšlenku návrhových vzorů pochází z roku 1977. Název „builder pattern“ se však v odborné literatuře před rokem 1995 a zveřejnění knihy návrhové vzory nevyskytuje.

Vzorem se dále zabývá Josua Bloch ve své knize Effective Java 5. Kdy navrhuje alternativu zaměřenou na odstranění opakování kódu v třídách Concrete Builder (viz Struktura).

Motivace užití

Použití Stavitele je vhodné u rozdílných tříd s podobným procesem konstrukce. Samotné třídy mohou být rozdílné a nezávislé. Tím se liší od Abstraktní továrny, která slouží k tvorbě podobných objektů. Podobných z hlediska prezentace v systému Windows, nebo Mac. Důvodem pro použití je oddělení vytváření složitých objektů od jejich prezentace, které by mělo vést k přehlednějšímu kódu.

Struktura



Obrázek 3 .4: *Návrhový vzor Builder*

Director - Třída řídící proces vytváření objektů.

„Instance třídy Director získává od klienta určení, který Builder má použít. Tím je i definováno, který produkt bude vyráběn. Director řídí vytvoření produktu voláním metod pro zhotovení jednotlivých částí z rozhraní Builder. Po vytvoření požadované části výsledného produktu je tato do něho začleněna. Výsledný produkt je možné získat použitím metody getResult. Director je odstíněn od způsobu, jakým se vytvářejí konkrétní části a jak se skládají. Určuje, ovšem kdy se mají části vyrobit a tím řídí proces vytváření produktů.“

Builder

Abstraktní rozhraní pro tvorbu objektů (produkt). Metoda buildPart slouží k vytvoření konkrétní části objektu, o její volání se stará Director.

Concrete Builder

Implementace rozhraní Builder schopná vytvářet další objekty. Vytváří a sestavuje součásti pro tvorbu složitých objektů. Metoda getResult slouží k předání výsledného produktu a volá ji zpravidla klient, který konstrukci objektu vyvolal.

„Která z možných tříd ConcreteBuilder je použita pro vytvoření objektu, může být rozhodnuto na základě použitých dat, která mají být zobrazena. Například pokud má být zobrazeno několik

položek, je klientu předán jako Product seznam tvořen „checkBoxy“. V případě více položek bude vytvořen a předán rozbalovací seznam. Samozřejmě kromě zobrazovaných dat mohou hrát roli i jiné faktory. Directoru je předán jako parametr instance ConcreteBuilder a Director, s využitím metod nadefinovaných v rozhraní, řídí proces vytváření produktu.“

Příklad užití

Konstrukci provádí konkrétní implementace rozhraní Builder, ale proces konstrukce řidí třída Director. Například klient přijde k automatu a chce kapučíno. O tvorbu kapučína se stará automat (direktor), klient si zvolí nápoj stiskem tlačítka. Automat provádí vždy ty samé kroky, existuje tedy jediné výrobní schéma, ale podle stisku tlačítka mohou být některé kroky vynechány (cukr), některé vzájemně zaměnitelné (kapučíno vs. čaj do kelímku). O realizaci těchto jednotlivých kroků se starají různé implementace třídy Builder (Stavitel čajů, stavitel kafí, apod.). Nápoj je možné odebrat teprve poté, kdy jsou všechny kroky konstrukce dokončeny.

Tedy zjednodušeně: Ke konstrukci jednoho objektu slouží právě jeden Builder, proces konstrukce řídí Direktor, objekt ke konstrukci vybírá klient předáním Builderu Direktoru, po ukončení konstrukce si z Builderu pouze vybere výsledek metodou getResult.

Stavitelů ke konstrukci jednoho objektu může být voláno i více. Například pokud je výsledný produkt Composite může být při jeho konstrukci voláno více stavitelů, případně stejný stavitel několikrát za sebou. Tak je zajištěno například vytváření složitých objektů grafického rozhraní operačního systému.

Výsledek užití

„Je vytvořen princip konstrukce objektů, který je variabilní na základě používaných dat nebo nadefinování věcné logiky. Klient je odstíněn od tvorby objektů a pouze prezentuje výsledek.“

Užitečné typy při užití

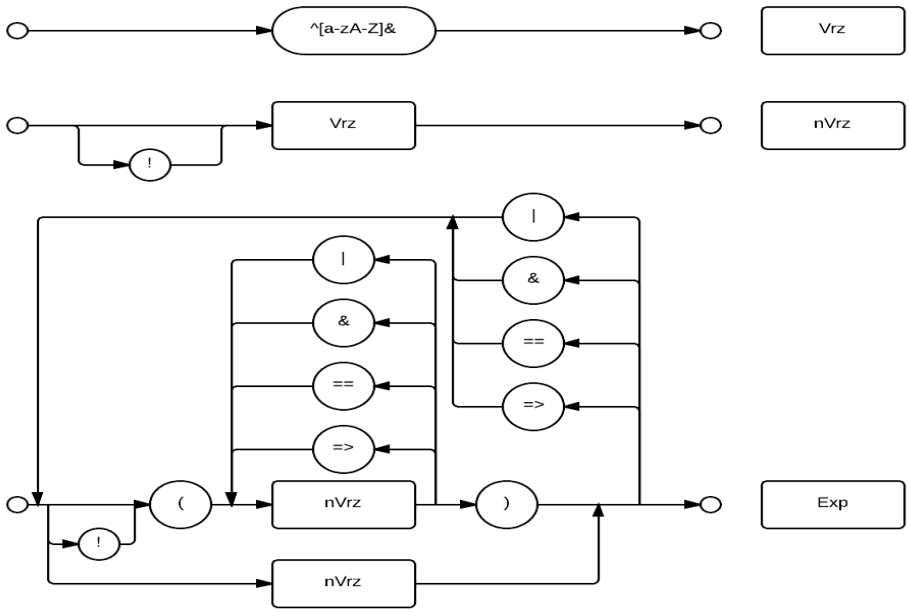
Builder slouží ke konstrukci složitých objektů po jednotlivých krocích. Zatímco Abstraktní továrna se používá při tvorbě blízké skupiny objektů (jednoduchých i komplexních). Abstraktní továrna vrací produkt v jednom kroku, u vzoru Builder je výsledný objekt vrácen po ukončení a jako výsledek konstrukce.

Návrhový vzor Stavitel se obvykle používá k tvorbě objektů typu Kompozit.

Zdroj:[4][5]

V mé aplikaci bude využit návrhový vzor Builder pro výstavbu stromové struktury, která se řídí návrhovým vzorem Kompozit. V roli direktora je Parser, který se stará o rozdělení výrazu, následně si uloží zadaný logický výraz do pole charů a dále testuje zda, je znak vyhovující tzn. zda se jedná o logickou operaci (konjunkce, disjunkce, ekvivalence, implikace, negace) nebo o znak pro logickou proměnnou.

Nejdříve jsem chtěl, aby aplikace byla schopná pracovat s libovolným počtem výrokových proměnných, jak je znázorněno na výše uvedeném obrázku syntax diagramu parseru. Tento parser by dle tohoto diagramu příjmal znaky reprezentované regulárním výrazem (^[a-zA-Z]$ tj. všechna malá a všechna velká písmena). Avšak po dohodě s vedoucím práce jsem omezil počet výrokových proměnných na pět.



Obrázek 3 .5: *Syntax diagram parseru*

3.2 Platforma

Tato kapitola seznamuje s platformou vývojového prostředí NetBeans a prostředky zvolenými k implementaci aplikace. Pro vývoj jsem použil softwarovou platformu JavaFX, která vychází z klasického JavaSE.

3.2.1 JavaFX

JavaFX je softwarová platforma na bázi Javy z dílny Sun Microsystems. pro vytváření a poskytování RIA aplikací (v překladu bohaté internetové aplikace), které mohou být spuštěny napříč širokým spektrem zařízení. Aktuální verze obsahuje podporu pro stolní počítače a webové prohlížeče pro Windows, Linux a Mac OS X. Od verze JavaFX 2.0 a novější je implementována jako nativní knihovna Java, a proto aplikace využívající JavaFX jsou psány v nativním kódu v jazyka Java. Pro stolní počítače, aktuální verze podporuje systémy Windows XP, Windows Vista, Windows 7, Mac OS X a Linux operační systémy.

3.2.2 Technické přednosti:

JavaFX 1.1 byla založena na konceptu "společného profilu", který je určen k rozpětí napříč všemi zařízeními s podporou JavaFX. Tento přístup umožňuje vývojářům používat společný programovací model při vytváření aplikace cílené jak pro stolní počítače, tak pro mobilní zařízení a sdílet hodně kódu, grafických aktiv a obsahu mezi stolní a mobilní verzí. Při potřebě ladění aplikací na určitou třídu zařízení, JavaFX 1.1 obsahuje platformová API, které jsou pro stolní či mobilní aplikace specifické. Například JavaFX Desktop profil obsahuje Swing a pokročilé vizuální efekty.

Drag-to-Install. Z pohledu koncového uživatele funkce "Drag-to-install" umožňuje přetáhnout JavaFX widget a umístět jej do svého počítače. Aplikace neztratí svůj stav nebo kontext i po zavření prohlížeče. Aplikace může být rovněž znovu spuštěna kliknutím na zástupce, který se vytvoří automaticky na pracovní ploše uživatele.

3.2.3 Platforma JavaFX 2.x zahrnuje následující komponenty:

NetBeans IDE pro JavaFX: NetBeans s drag-and-drop paletami přidává objekty s efekty transformace, animace a sady vzorků. Pro podporu JavaFX 2 je potřeba alespoň NetBeans 7.1.1. Pro uživatele Eclipse je JavaFX podporovaná zásuvným modulem umístěným v projektu Kenai.

Java FX Scene Builder: Tento nástroj byl zaveden pro JavaFX 2.1 a novější. Přetažením palety je vytvořeno uživatelské rozhraní, které je uloženo jako FXML soubor. Programátor pomocí Java FX Scene Builderu může jednoduše a pohodlně vytvářet uživatelská rozhraní.

3.2.4 Historie

JavaFX Script, která je součástí skriptování JavaFX, začala existovat jako projekt Chrise Olivera s názvem F3. Společnost Sun Microsystems oznámila první JavaFX na konferenci JavaOne Worldwide Java Developer v květnu 2007.

V květnu 2008 společnost Sun Microsystems oznámila plány poskytnout JavuFX pro počítače, pro mobilní zařízení, ve druhém čtvrtletí roku 2009. Od konce července 2008, vývojáři mohli stáhnout náhled JavaFX SDK pro Windows a Macintosh a také zásuvný modul pro NetBeans JavaFX 6.1.

JavaFX 1.0
Dne 04.12.2008 vydal Sun Microsystems JavaFX 1.0.

JavaFX 1.1
JavaFX pro mobilní zařízení. Vývoj byl k dispozici jako součást verze JavaFX 1.1. Oficiální oznámení 12. února 2009.

JavaFX 1.2

JavaFX 1.2 byla vypuštěna na dnech JavaOne 2. června 2009. Tato verze představila:

Beta podpora pro Linux a Solaris.

JavaFX vstupní/výstupní řízení, maskování rozdílů mezi stolní a mobilní aplikací.

Zvýšení rychlosti.

Windows Mobile Runtime s Sun Java Wi-Fi klientem.

JavaFX 1.3

JavaFX 1.3 byla vydána 22. dubna 2010. Tato verze představila:

Zlepšení výkonu.

Podpora dalších platforem.

Vylepšená podpora pro ovládací prvky uživatelského rozhraní.

JavaFX 1.3.1

Tato verze byla vydána 21. srpna 2010. Tato verze představila:

Rychlý start aplikací JavaFX.

Vlastní načítací lištu pro spuštění aplikace.

JavaFX 2.0

Tato verze byla vydána 10. října 2011. Verze představila:

Novou sadu rozhraní Java API. JavaFX otevřela možnosti pro všechny Java vývojáře, aniž by bylo nutné je učit nový skriptovací jazyk. Podpora pro Java Script FX byla vynechána trvale.

Ukončení podpory pro JavaFX Mobile.

Oracle oznámil svůj záměr open source JavaFX.

Různá vylepšení byly provedeny v rámci knihovny JavaFX pro vláknové operace (multithreading). Dotazová API byla aktualizována na podporu mnohem výstižnějších vláknových operací (tj. třída JavaTaskBase již není nutná).

JavaFX 2.0 Beta

26. května 2011, Oracle vydal JavaFX 2.0 Beta.

Beta verze byla k dispozici pouze pro 32 a 64 bitové verze systému Microsoft Windows XP, Windows Vista a Windows 7. JavaFX 2.0 byla vydána pouze s podporou Windows. Podpora pro Mac OS X byla přidána v verzi JavaFX 2.1. Linuxu byla přidána podpora s verzí JavaFX 2.2.
JavaFX 2.0 využívá nový deklarativní jazyk XML s názvem FXML.

JavaFX 2.1

Dne 27. dubna 2012, Oracle vydal verzi 2.1, která obsahuje následující hlavní rysy:

- První oficiální verze pro Mac OS X (pouze stolní počítače).
- Vylepšení uživatelského rozhraní např. grafy (skládaná grafy) a panely menu.
- Webview komponenta nyní umožňuje volat JavaScript na metody Javy.

JavaFX 2.2

Tato verze byla vydána 14. srpna 2012 obsahuje následující hlavní rysy:

- Podporu pro Linux (včetně podpory WEBSTART).
- Podporu pro HTTP živé přehrávání (livestreaming).
- Obrázek manipulace API.
- Nativní balení.

JavaFX 2.2 přidává nové balení s názvem Nativní balení, což umožňuje balení aplikace jako "nativní svazek". To dává uživatelům možnost instalovat a spustit aplikaci bez jakýchkoli externích závislostí na systému JRE nebo FX SDK.

3.2.5 Licence

V současné době existuje několik licencí pro moduly, z kterých se skládá JavaFX:

- Část jádra JavaFX je stále proprietární software a jeho kód nebyl dosud zveřejněn,
- JavaFX překladač a starší verze 2D grafů jsou uvolněny pod licencí GPL v2,
- Plugin JavaFX pro NetBeans je dvojí pod licencí GPL v2 a CDDL.

Na JavaOne 2011, Oracle Corporation oznámila, že JavaFX 2.0 se stane open source. Od prosince 2011 Oracle začal otevírat zdrojový kód JavaFX pod GPL licencí.

Zdroj: [6]

3.2.6 NetBeans

NetBeans je open source projekt s rozsáhlou uživatelskou základnou, komunitou vývojářů a s víc než 100 partnery po celém světě. Pod firmu byl Sun Microsystems, která je hlavním sponzorem projektu, přešel na základě akvizice stejnojmenné české společnosti v říjnu 1999. Pod open-source licencí byl produkt uvolněn v červnu 2000.

Vývojové prostředí NetBeans IDE je nástroj, pomocí kterého programátoři mohou psát, překládat, ladit a šířit programy. NetBeans IDE je napsáno v jazyce Java a je postaveno na stejnojmenné platformě. Primárně je určeno pro vývoj aplikací v jazyce Java, ale může podporovat i další programovací jazyky (ve verzi 6.0 např. C++, PHP, Ruby). V Javě podporuje všechny 3 hlavní platformy - J2SE, J2EE a J2ME. Zjednodušuje práci s frameworky jako je Struts nebo Ruby on Rails a umožňuje také vývoj webových služeb a webových aplikací. Obsahuje také nástroje pro tvorbu

designu aplikace. Kromě toho také existuje velké množství modulů, které toto vývojové prostředí rozšiřují. Vývojové prostředí NetBeans je bezplatně šířený produkt, který je možné používat bez jakýchkoliv omezení. Je naprogramovaný v Javě, takže jej lze spustit na operačních systémech Windows, Linux, Mac OS X a Solaris.

Zdroj: [7]

3.3 Realizace

V této kapitole je představena implementace aplikace, která je cílem této bakalářské práce. Představím zde aplikaci jak z uživatelského hlediska, tak z hlediska vývojového.

3.3.1 Aplikace na straně klienta

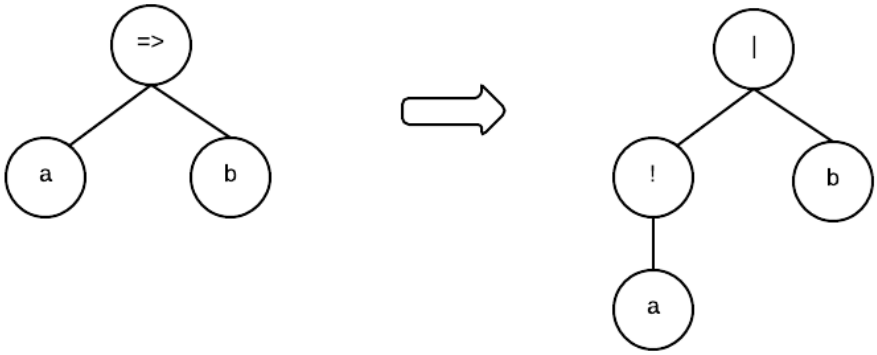
Uživatelská část aplikace byla naimplementována ve vývojovém prostředí NetBeans 7.2.1 za použití technologie Java FX verze 2.2. Vývojové prostředí NetBeans jsem zvolil, protože obsahuje plugin pro vývoj JavaFX aplikací, který nabízí nástroje jako automatický náhled vzhledu okna, protože vývojové prostředí NetBeans automaticky komunikuje s JavaFX Scene Builderem, v kterém se uživatelské rozhraní vytváří, nebo JavaFX kompilátor. Jako hlavní zdroj při získávání informací o technologii JavaFX jsem použil zejména oficiální dokumentaci k JavaFX.

K spouštění aplikace na straně klienta stačí, aby měl klient pouze nainstalovanou aktuální verzi Java (Java 7 a vyšší). Dál už není potřeba žádných dalších balíčků, nebo pluginů. Po zkompilování aplikace JavaFX kompilátorem, můžeme aplikaci spouštět jako webovou i konzolovou.

3.3.2 Převod do konjunktivní normální formy

Pro správné vykonání rezoluční metody je potřeba převést úsudek do konjunktivní normální formy(KNF). V aplikaci používám pro převod následující pravidla:

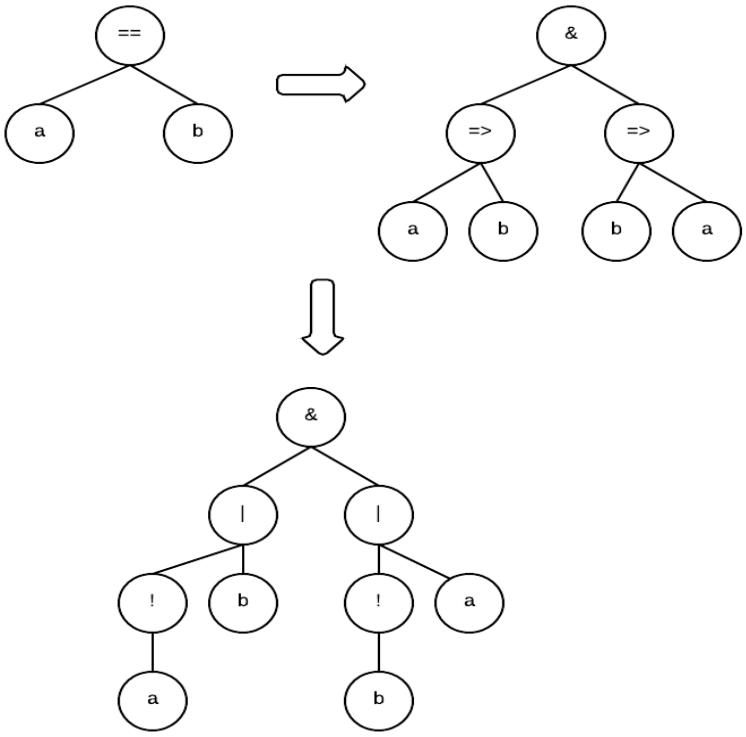
Převod implikace:



Obrázek 3 .6: *Diagram pro převod implikace do KNF*

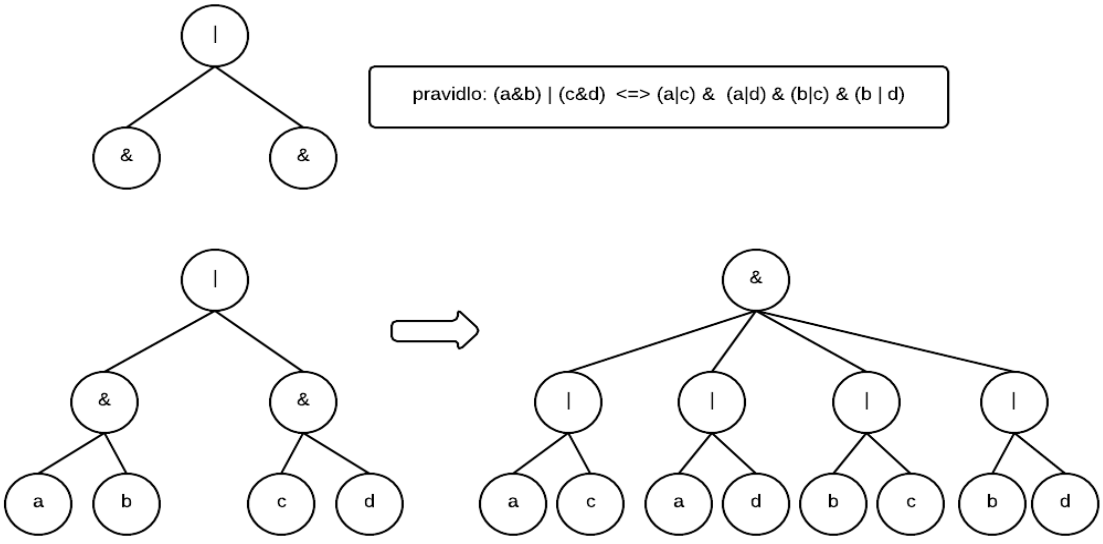
Převod ekvivalence:

Ekvivalenci můžeme rozdělit jako dvě implikace, proto bude převod vypadat následovně:



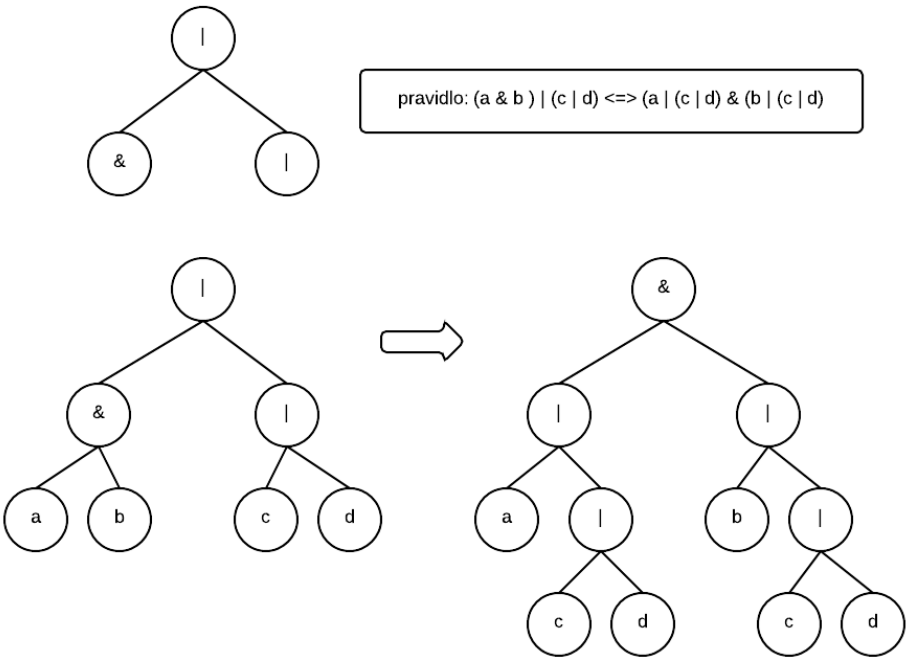
Obrázek 3 .7: *Převod ekvicalence do KNF*

Další pravidla, pro převod do KNF



Obrázek 3 .8: *Pravidlo pro převod do KNF*

Pravidlo pro převod do KNF



Obrázek 3 .9: *Pravidlo pro převod do KNF*

3.3.3 **Balíčky aplikace**

Kód jsem rozdělil do několika balíčků, pro jeho přehlednost a snadnou orientaci.

- cz.vsb.fei.proofs
- cz.vsb.fei.proofs.builder
- cz.vsb.fei.proofs.controllers
- cz.vsb.fei.proofs.di
- cz.vsb.fei.proofs.di.annotations
- cz.vsb.fei.proofs.di.listeners
- cz.vsb.fei.proofs.elements
- cz.vsb.fei.proofs.exceptions
- cz.vsb.fei.proofs.iterators
- cz.vsb.fei.proofs.parsing
- cz.vsb.fei.proofs.resources
- cz.vsb.fei.proofs.resources.css
- cz.vsb.fei.proofs.resources.fxml
- cz.vsb.fei.proofs.resources.images
- cz.vsb.fei.proofs.solvers.contradiction
- cz.vsb.fei.proofs.solvers.deduction

- cz.vsb.fei.proofs.solvers.deduction.rules
- cz.vsb.fei.proofs.solvers.hilbert
- cz.vsb.fei.proofs.structure
- cz.vsb.fei.proofs.ui

Balíček cz.vsb.fei.proofs

Balíček s main třídou, v které se nastavuje velikost a parametry zobrazovaného okna.

Balíček cz.vsb.fei.proofs.builder

Obsahuje interface a implementaci builderu 0 (používá se při parsování), parser posílá builderu příkazy a builder vytvoří datovou strukturu logického výrazu.

Balíček cz.vsb.fei.proofs.controllers

V tomto balíčku kontroluji a zajišťuji komunikaci mezi uživatelským rozhraním a ostatními třídami a metodami JavaFX. Při volání objektů z uživatelského rozhraní musím používat anotaci @FXML. Pro každý objekt vytvořený v JavaFX Scene Builderu (např. tlačítko) zde nastavím co zvolený objekt má vykonávat. Pro každou záložku aplikace je jeden controller , jeden hlavní.

Balíček cz.vsb.fei.proofs.di

Třídy týkající se Dependency Injection

Balíček cz.vsb.fei.proofs.di.annotations

Anotace používané pro Dependency Injection

Balíček cz.vsb.fei.proofs.di.listeners

Listenery pro vlastní injektování

Balíček cz.vsb.fei.proofs.elements

V tomto balíčku řeším nevýhodu návrhového vzoru kompozit a to tu, že jednotliví členové musí implementovat metodu ToString(). V mém případě je to metoda stringValue(), která vrací pouze řetězec s znakem, který interpretuje danou logickou operaci. Pro názornost přikládám metodu pro konjunkci, z které je použití metody stringValue() názorné Každý uzel stromu má některý z těchto elementů.

```
public class AndOperation implements ExpressionElement
{
	@Override
	public String stringValue()
	{
		return "&";
	}
}
```

Balíček cz.vsb.fei.proofs.exceptions

Balíček obsahující výjimky, které mohou v programu nastat. Jedná se zejména o SyntaxException.

Balíček cz.vsb.fei.proofs.iterators

Tento balíček obsahuje iterátory pro různé průchody datovou strukturou logického výrazu. Zajišťují procházení prvků, které jsou uloženy v složité datové struktuře. Samotná implementace iterátoru je před jeho uživatelem skryta.

Balíček cz.vsb.fei.proofs.parsing

Obsahuje parser, který čte text zadaný uživatelem do TextFieldu po znacích a posílá příkazy builderu.

Balíček cz.vsb.fei.proofs.resources

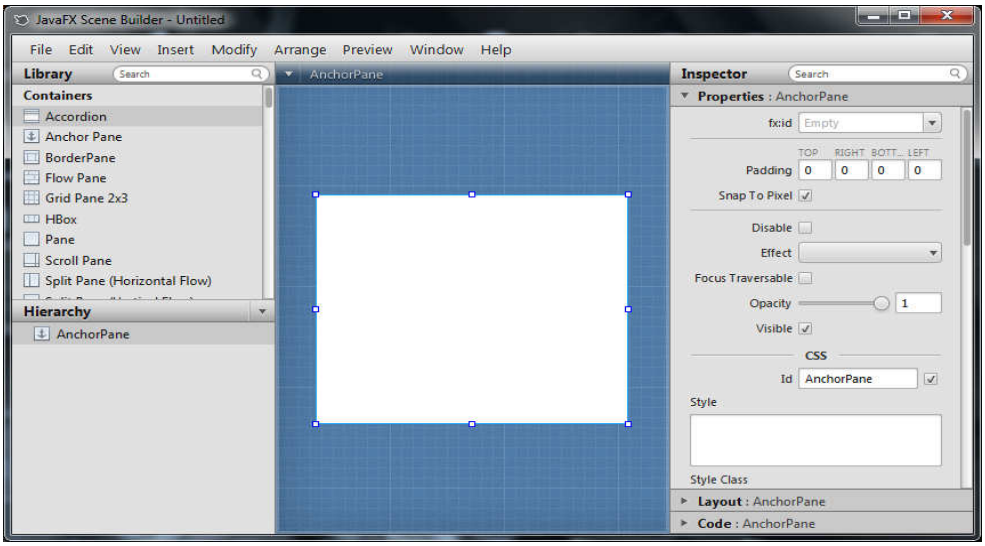
V tomto balíčku jsou doplňující zdroje, obsahuje třídu pro načítání zdrojů.

Balíček cz.vsb.fei.proofs.resources.css

Obsahuje CSS pro jednotlivé záložky aplikace, pomocí těchto souborů se formátuje např. velikost písma a podbarvení sloupců v tabulce.

Balíček cz.vsb.fei.proofs.resources.fxml

V tomto balíčku jsou soboru nutné k zobrazení a funkčnosti uživatelského rozhraní. Nalezneme zde .fxml soubory, v kterých je uloženo uživatelské rozhraní, které můžeme měnit pomocí JavaFX Scene Builderu, který je nutno doinstalovat zvlášť, poté ale funguje harmonicky s vývojovým prostředím NetBeans.



Obrázek 3 .10: *JavaFX Scene Builder*

Balíček cz.vsb.fei.proofs.resources.images

Zde jsou umístěny obrázky, které aplikace využívá.

Balíček cz.vsb.fei.proofs.solvers.contradiction

V tomto balíčku jsou třídy, které jsou zodpovědné za řešení sporem.

Balíček cz.vsb.fei.proofs.solvers.deduction

Obsahuje třídy zodpovědné za řešení pomocí přirozené dedukce, starají se o aplikování dedukčních pravidel.

Balíček cz.vsb.fei.proofs.solvers.deduction.rules

Obsahuje třídy, které se používají pro transformaci výrazů tzn. pravidla, která se aplikují na logické formule.

Balíček cz.vsb.fei.proofs.solvers.hilbert

V tomto balíčku jsou třídy, které řeší metodu Hilbertova principu, starají se o aplikování pravidel a axiomů.

Balíček cz.vsb.fei.proofs.structure

V tomto balíčku je uložena reprezentace stromu, která se řídí návrhovým vzorem kompozit, který je zmíněn v kapitole 3.1. Kompozit je zde reprezentován rozhraním ExpressionNode. Dále zde jsou třídy pro práci s logickými operacemi: AndOperator (Konjunkce), OrOperator (Disjunkce), EquivalenceOperator (Ekvivalence), ImplicationOperator (Implikace) a UnaryOperator (Negace). Tyto třídy jsou uzly ve stromu, pro interpretaci listů stromu slouží třída LogicOperand.

Každá z těchto metod obsahuje konstruktor, v kterém nastavuji hodnoty prvního a druhého operandu. Následuje metoda evaluate, která řeší samotnou logickou operaci dvou logických operandů. Dále se nastavuje priorita provádění logického výrazu. Podle které se operace umisťuje do stromu. Následuje metoda getExpressionElement, která odkazuje do balíčku cz.vsb.fei.proofs.elements.

Balíček cz.vsb.fei.proofs.ui

Zde se nachází třídy, které jsou nutné pro vykreslování některých prvků uživatelského rozhraní, jako např. řádků v tabulce.

3.3.4 Unit testovaní

Nazývané také jako jednotkové testování. Unit testování se zakládá na testování nejmenších částí systému nezávisle na ostatních částech. Existuje několik frameworku na podporu testování. Pro tento projekt byl vybrán velmi rozšířený a dobře podporovaný framework a to JUnit. Testy v JUnit vypadají velmi podobně jako obyčejné Java třídy. Hlavním nástrojem pro testování jsou assert metody. Pro splnění testu, musí všechny asserty v testovací metodě bezproblémově projít. Pokud i jediný selže, selhal celý test. Existuje několik druhu assertu, jak naznačuje tabulka.

Metoda	Účel
assertTrue(boolean condition)	Vyhodnotí se jako správná, pokud se condition vyhodnotí jako true.
assertFalse(boolean condition)	Vyhodnotí se jako správná, pokud se condition vyhodnotí jako false.
assertArrayEquals(Object[] expected, Object[] actual)	Slouží k porovnání dvou objektů. Vyhodnotí se jako správná pokud jsou obě pole shodná.
assertEquals(Object expected, Object actual)	Porovná dva libovolné objekty. Pokud jsou oba objekty shodné, vyhodnotí se jako správná.
assertNotNull(Object object)	Pokud objekt není null, vyhodnotí se jako správná.
assertThat(T actual, Matcher <T> matcher)	Do této metody lze dodat vlastní implementace porovnacího algoritmu (Matcher)

Tabulka.3.2: Typy metod

Další nedílnou součástí test case jsou anotace JUnit. Ty naznačují JUnit frameworku jak má zacházet s danými metodami. Nejdůležitější anotace shrnuje tabulka.

Anotace	Význam
@Test	Definuje metodu, která je brána jako test. Může obsahovat více assertů, které musí být splněny najednou, aby byl celý test splněn. Jedna třída může obsahovat více metod s touto anotací.
@Before	Anotace používaná k označení metody, která se zavolá před započetím testu. Je možné ji využít pro vytvoření kontextu nebo např. k získání původních dat.
@After	Označuje metodu, která se zavolá po dokončení testu. Vhodná např. k uklizení kontextu nebo navrácení původních hodnot.

Tabulka.3.3: Anotace JUnit

Ukázku jedné metody využité pro testování ukazuje výpis . V této testovací metodě (nese anotaci @Test) se testuje metoda pro ekvivalenci. Pro svůj test využívám anotaci assertEquals. Do dvou objektů rozhraní ExpressionNode zadám logický operand s hodnotami „pravda“ a „pravda“.

Ekvivalence pro tyto dvě hodnoty je pravdivá, takže do očekávaného výsledku zadávám také hodnotu „pravda“ a po porovnání těchto zadaných hodnot ověřím, že funkce pro Ekvivalenci výrazu vrací správnou hodnotu. Tento výpočet proběhl v pořádku a tak je celý test tedy úspěšný.

```
@Test
    public void trueTrue() {
       ExpressionNode firstOperand;
       firstOperand = new LogicOperand("true");
       ExpressionNode secondOperand;
       secondOperand = new LogicOperand("true");
       ExpressionNode EquivalenceOperation = new
EquivalenceOperator(firstOperand,secondOperand);
       Boolean expected = new Boolean("true");
       assertEquals(expected, EquivalenceOperation.getValue());
    }
```

Tyto testy jsem udělal pro každou používanou metodu (tj. negace, konjunkce, disjunkce, ekvivalence a implikace). Ve všech binárních kombinacích těchto metod. Dále jsem jednotkových testů využil u Builderu, kde jsem vyzkoušel, že výstavba výrazu funguje správě.

Zdroj: [8][9]

3.3.5 Uživatelská příručka

Po spuštění aplikace se uživateli zobrazí hlavní formulář aplikace, který vypadá takto:

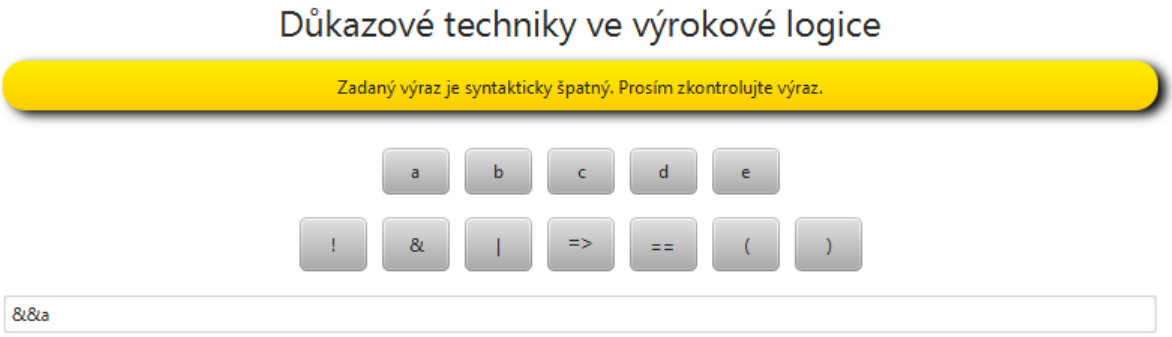
Obrázek 3 .11: *Hlavní formulář aplikace*

Uživatel zadává předpoklady do připravené kolonky (červená šipka na výše zobrazeném obrázku) a tlačítkem „Přidat předpoklad“ se předpoklad přidá do tabulky předpokladů. Po přidání předpokladů, uživatel napíše závěr a za pomoci tlačítka „Přidat závěr“ se závěr přidá do pole připraveného pole pro závěr.

Uživatel může do připraveného pole pro zadávaní psát výrokové formule ručně, nebo za pomoci tlačítek, které jsou uvedeny pod názvem aplikace. Výrokové formule nemusí být striktně pojmenovány a,b,c,d,e, ale uživatel si může zvolit jaké pojmenování uzná za vhodné, avšak musí tyto formule zadávat z klávesnice. Pro názornost tlačítek s výrokovými funkcemi (negace, konjunkce, disjunkce, implikace, ekvivalence) je při najetí na tlačítko zobrazen jeho význam.

Obrázek 3 .12: *Tlačítka s výrokovými funkcemi*

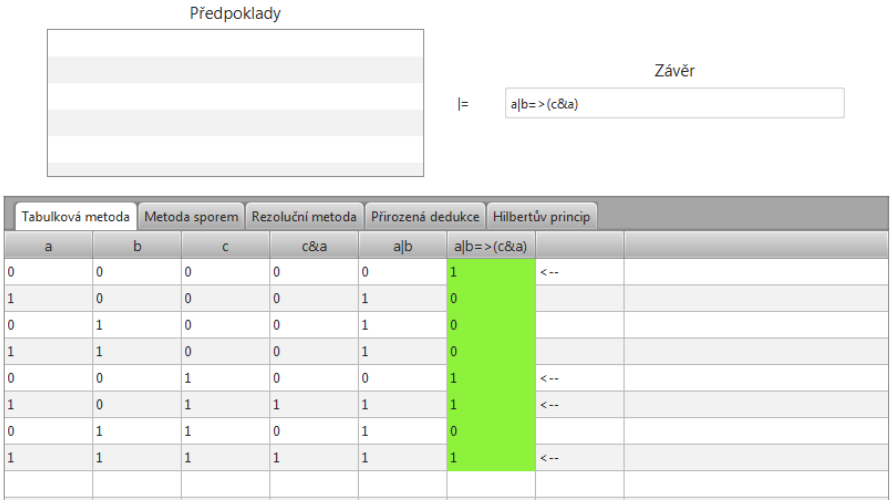
Po zadání formule parser zpracuje výraz a pokud je v pořádku, tak jej přidá do tabulky předpokladů. Pokud ne, vyskočí varovná hláška.



Obrázek 3 .13: *Ukázka špatně zadaného výrazu*

Parser dokáže zpracovávat výrazy zadané jak korektním způsobem (např. a=>b), tak si umí poradit také s nesprávně zadanou formulí, avšak musí být dodržen počet výrokových proměnných a výrokových funkcí (např. => a b, tento výraz parser zpracuje jako a=>b).

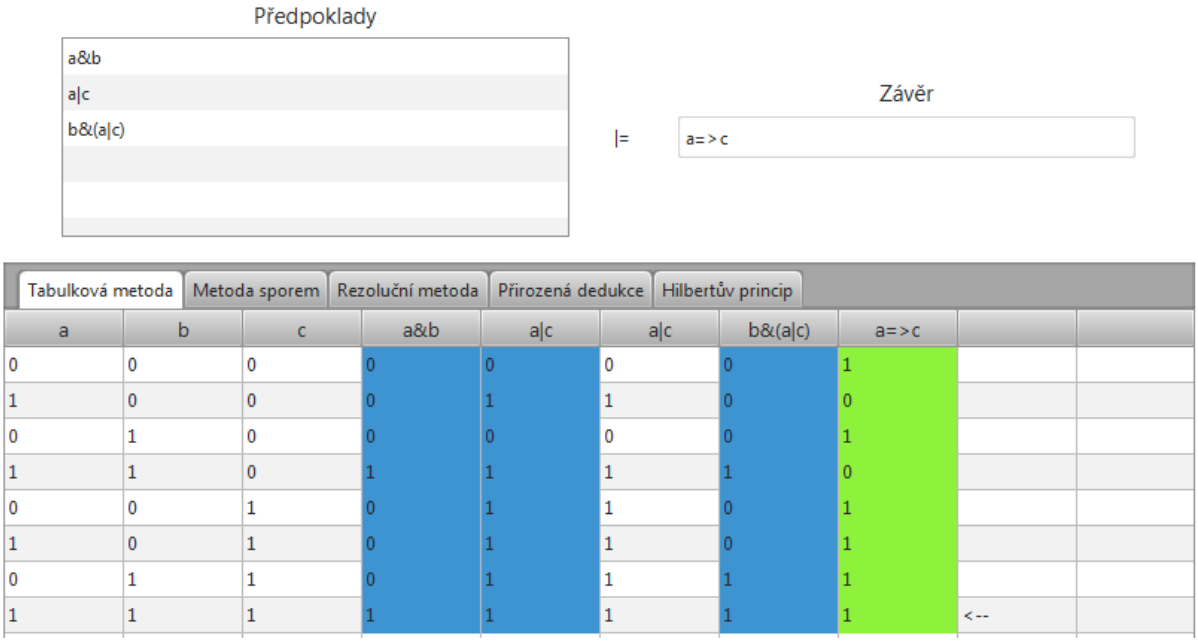
Pokud chceme použít tabulkovou metodu pouze pro jednu výrokovou formuli (tzn. nebudeme zadávat předpoklady a závěr), tak tuto formuli uživatel zadá jako závěr a aplikace vyhodnotí pouze tabulkovou metodu. Formule, kterou zadal uživatel, je označena v tabulce zelenou barvou. Pro ohodnocení v kterém je kladná je označena šipkou.



a	b	c	c&a	a|b	a|b=>(c&a)	
0	0	0	0	0	1	<--
1	0	0	0	1	0	
0	1	0	0	1	0	
1	1	0	0	1	0	
0	0	1	0	0	1	<--
1	0	1	1	1	1	<--
0	1	1	0	1	0	
1	1	1	1	1	1	<--

Obrázek 3 .14: *Ukázka tabulky pro jednu výrokovou formuli*

Při zadávání předpokladů a závěru jsou předpoklady vyobrazeny v tabulce modrou barvou a závěr zelenou. Pro ohodnocení, kde jsou předpoklady pravdivé a závěr také je vyznačena šipka.



a	b	c	a&b	a|c	a|c	b&(a|c)	a=>c		
0	0	0	0	0	0	0	1		
1	0	0	0	1	1	0	0		
0	1	0	0	0	0	0	1		
1	1	0	1	1	1	1	0		
0	0	1	0	1	1	0	1		
1	0	1	0	1	1	0	1		
0	1	1	0	1	1	1	1		
1	1	1	1	1	1	1	1	<--	

Obrázek 3 .15: *Ukázka tabulky včetně předpokladů a závěru*

Tlačítko smazat předpoklad smaže z tabulky předpokladů právě označený předpoklad. Tlačítko smazat vše vymaže obsah celého formuláře.

4 Závěr

Hlavním cílem mé bakalářské práce bylo vytvořit aplikaci, která provádí dokazování platnosti úsudku výrokové logiky. Jedná se o tabulkovou metodu, metodu sémantického sporu, rezoluční metodu, přirozenou dedukci a Hilbertův systém. To se mi podařilo a aplikace je funkční, alespoň na testovaných příkladech. Nejsem si zcela jist, zda metoda Hilbertovým systémem bude funkční pro všechny příklady, ale bohužel jsem nepřišel na lepší řešení, jak tuto metodu lépe vyřešit.

Při tvorbě aplikace mi největší problémy dělal již zmiňovaný Hilbertův systém, dále jsem se potýkal s problémy při převodu formulí do konjunktivní normální formy, avšak s tímto problémem jsem si poradil.

Systém by měl sloužit jako výukový materiál pro výuku dokazování ve výrokové logice a je umístněn na reálném serveru http://dt.a-fw.net, kde je volně dostupný. Dále v práci zmiňuji teorii k dokazování výše zmíněných metod. Díky výhodě platformy JavaFX jsem zhotovil také konzolovou aplikaci.

Směr kterým by se aplikace mohla rozvíjet dále je podle mne hlavně v vyřešení problému s Hilbertovým systémem. Dále by aplikace mohla podporovat volbu jazyka a být propojena s databází, kde by uživatel po přihlášení viděl své naposled zadávané formule. Registrace by však byla dobrovolná, aby systém bylo možno dále používat jednorázově, bez nutnosti registrace.

Použitá literatura

[1] DUŽÍ, Marie. *Logika pro informatiky (a příbuzné obory): učební text*. 1. vyd. Ostrava: VŠB-TU Ostrava, 2012, 179 s. ISBN 978-802-4826-622.

[2] SAWA, Zdeněk. *Úvod do teoretické informatiky: logika, množiny, matematická notace* [online]. 2013 [cit. 2013-04-22]. Dostupné z: http://www.cs.vsb.cz/sawa/uti/materialy/uti-1.pdf

[3] VONDRÁK, Ivo. *Úvod do softwarového inženýrství* [online]. Ostrava, 2002 [cit. 2013-07-11]. Dostupné z: http://vondrak.cs.vsb.cz/download/Uvod_do_softwaroveho_inzenyrstvi.pdf

[4] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4

[5] . Builder. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-04-22]. Dostupné z: http://cs.wikipedia.org/wiki/Builder

[6] JavaFX. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-04-22]. Dostupné z: http://en.wikipedia.org/wiki/JavaFX

[7] NetBeans. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-07-11]. Dostupné z: http://cs.wikipedia.org/wiki/Netbeans.

[8] MALÝ, Martin. Ještě k testování. In: *Www.zdrojak.cz* [online]. [cit. 2013-04-05]. Dostupné z: http://www.zdrojak.cz/clanky/jeste-k-testovani/

[9] Programování řízené testy. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2013-04-05]. Dostupné z: http://cs.wikipedia.org/wiki/Programov%C3%A1n%C3%AD_%C5%99%C3%ADzen%C3%A9_testy

Seznam příloh

Součástí BP/DP je CD/DVD.